(2)

ETL-SR-7

PDEF: A Standard File Format For Data Interchange

E T L

Michael M. McDonnell

January 1991

U.S. Army Corps of Engineers
Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

U.S. ARMY ENGINEER TOPOGRAPHIC LABORATORIES
ETL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE January 1991	3. REPORT TYPE AND DATES COVERED Final: 1 June 1989 - 1 June 1990

4. TITLE AND SUBTITLE
PDEF: A Standard File Format for Data Interchange

5. FUNDING NUMBERS
PR: 4A161102B52C
TA: C0
WU: 014

6. AUTHOR(S)
Michael M. McDonnell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

8. PERFORMING ORGANIZATION REPORT NUMBER
ETL-SR-7

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This report explains a new method of encoding data in a set of files that allows advantages over current formats and methods. A new format is necessary because all current widely-available file formats retain restrictions (such as fixed field lengths) that are no longer necessary with modern programming languages. In particular, the ability to be parsed by FORTRAN programs was formerly an important requirement. The limitations of FORTRAN lead to file formats that are clumsy and difficult to work with. The proposed format has a simple syntax and flexible semantics. The format is efficient both for computers and people. It is efficient for computers because it makes use of the powerful parsing tools available for the C programming language. It is efficient for people because data descriptions are in a human-readable form and the content of a data file can be understood without a user's manual. The bulk of this report is in appendixes which give illustrative examples Examples are presented for raster, quadtree, and vector data formats since these formats are especially prevalent in spatial data systems, a specialty of the U.S. Army Engineer Topographic Laboratories. PDEF stands for Protean Data Exchange Format.

14. SUBJECT TERMS Geographic Information System, Data Interchange, Spatial Data Storage, Data Exchange, Reformatting.

15. NUMBER OF PAGES
54

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

CONTENTS

PREFACE

This report was written at the request of Mr. Homer Babcock, Chief Geographic Information Systems Branch, Geographic Systems laboratory, U.S. Army Engineer Topographic Laboratories (ETL). The work was done in the ETL Research Institute, Center for Artificial Intelligence under DA Project 4A16110102B52C, TAsk CO, Work Unit 014, "Artificial Intelligence Concepts for Terrain Analysis."

The report was prepared between November 1989 and June 1990 under the supervision of Mr. John Benton and Mr. John Hansen, successive Team Leaders of the Center for Artificial Intelligence and of Dr. Richard Gomez, Director, Research Institute. Mr. Hansen's title was amended during this period to Chief, Artificial Intelligence Division.

Col. David F. Maune, EN, was Commander and Director, and Mr. Walter Boge was Technical Director of the U.S. Army Engineer Topographic Laboratories, Fort Belvoir, Virginia 22060-5546, during preparation of the report.

LIST OF ACRONYMS AND GLOSSARY

ADRG: Arc Digitized Raster Graphics; digital images of printed maps as produced by the U. S. Defense Mapping Agency.

ASCII: American Standard Code for Information Interchange; a conventional coding that represents alphanumeric characters by using numbers from 0 through 127.

C: A computer programming language.

FORTRAN: FORmula TRANslation; a computer programming language.

GIS: Geographic Information System. This is a computer program or a set of computer programs that allows the manipulation and display of spatial data and is used particularly for terrain data.

ISO: International Standards Organization, a European non-profit organization dedicated to the creation and promulgation of manufacturing standards.

Parse: In computer science this is the process of turning input data into data structures inside a computer.

PDEF: Protean Data Exchange Format.

Semantics: The context-dependent meaning of data. The semantics of data can be considered separately from the syntax.

SPOT: Systeme Probatoire d'Observation de la Terre. A satellite launched by the French national aerospace consortium (CNES) which makes images of the earth in three spectral bands.

Syntax: The form that data takes as opposed to the meaning that the data may have in some context.

XDR: eXternal Data Representation. This is a public-domain method of representing common types of computer data in a machine-independent form. XDR was designed and fostered by Sun Microsystems, Inc.

PDEF: A STANDARD FILE FORMAT FOR DATA INTERCHANGE

INTRODUCTION

The Protean Data Exchange Format (PDEF) is a set of computer programs which can be used to transform data between different systems which do not understand each other's formats. The problem of data transformation is not restricted to any particular discipline, but those of us concerned with digital terrain data have felt this problem acutely as we have found that different Geographic Information Systems (GIS) cannot use each other's data. Although PDEF was written to alleviate this particular problem, its uses are more general than the specific problem for which it was originally written. PDEF was designed primarily to be easy to use since reformatting decisions must be made by people. This report will examine both the format and the uses of PDEF.

This report is intended for both programmers and non-programmers who need to understand data formats and data format transform techniques. Programmers can find sufficient detail in the appendixes to implement PDEF. Programming examples are given there for various data types, such as raster, quadtree, and vector data. Non-programmers will find guidance in the body of the report on how to express their data-reformatting decisions in PDEF so that PDEF will reformat the data properly.

To understand PDEF, we need a brief review of data file formats. A *file* is a separate data entity on a disk or tape. It has its own name on a disk and is separated from other files on a tape by an end-of-file mark. Some files are further subdivided into records and fields. PDEF does not use any subdivisions finer than files. There are no records or prescribed fields defined in PDEF.

Data is commonly stored and transferred from one computer to another by using conventions in the formatting of the file, or files, containing data. It is common to have a *header* as the first part of a file. The header contains data about the file as a whole, such as the name of the data set, and also contains format information to help in reading the rest of the file. Headers typically contain a mixture of printable and binary data and are difficult to parse (i.e. understand) without a manual. There are no headers in PDEF. The rest of this introduction gives the rationale for the design of PDEF.

Designing a new data format (and implementing tools such as parsers to allow people to work with it) is a large undertaking and should only be done if there is a strong need. Is there, then, a good reason for defining PDEF? One reason is the influence of old formats and old languages such as FORTRAN. FORTRAN has dominated the design of file formats to date and causes many of the problems users have when attempting to understand and work with a current format.

FORTRAN cannot allocate data dynamically. This lack of dynamic allocation forces data file formats to have fixed field lengths, which causes some problems. For example, if a user wants to name a file using a title that is 20 letters long but the data file has only set aside 15 spaces for the name, then the desired filename must be shortened to a length of 15 letters or less. Carried to extremes, this leads to names for functions and variables that are restricted to a few upper-case ASCII characters and therefore have almost no semantic content. What does a function named SAXPY or QRTPE do? There is no way to tell without a manual. Similarly, and more to the point of this discussion of data formats, what data does field FTLLP contain? Again, there is no way to tell.

A problem encountered when dealing with many current data formats is a waste of space on data transfer media. If fixed-length fields have to be made large enough to contain the largest expected data element, then for an average data element there will be unused space that must be transferred on the tape anyway. Attempts to overcome this limitation, such as the ISO 8211 data transfer standard,[1] have a daunting complexity. The author has recently written a parser for ISO 8211 and it was a difficult task.

Another problem with most current formats is that they were designed by committees and therefore have a lack of conceptual integrity and an unnecessary complexity which is characteristic of such works. As an example, the proposed spatial data transfer specification which has been published by the Digital Cartographic Data Standards Task Force (DCDSTF) has a specification that is over 120 pages long.[2] Such specifications as DCDSTF are unworkably complex and will have to be changed later, leading to problems with versions of the "standard." In contrast, PDEF is simple. This simplicity is mostly a result of separating syntax from semantics, as will be illustrated later. Simplicity and human-readability were the guiding precepts in its design. However, simplicity must not be sacrificed to usability or usefulness. PDEF proves that a format can be both simple and useful.

A GENERAL DESCRIPTION OF PDEF

In PDEF a single data set is typically stored in several separate files, with (mostly) meta-data in those files that are readable by people. Meta-data is data about data. It includes such information as where the data is found, how many bits there are per data element, and whether data is to be read as a string of characters, as an integer, as an array of 32-bit floating-point numbers, etc. Although most current file formats contain such information in a file "header," PDEF has no file headers. In PDEF, a separate, human-readable file contains the meta-data typically found in headers. The bulk of the data is then placed in another file or set of files that contain nothing but binary data. Binary data files have the following characteristics:

[1] *Information Processing - Specification for a data descriptive file for information interchange*, International Organization for Standardization publication ISO 8211-1985(E), 15 Dec 1985.

[2] Digital Cartographic Data Standards Task Force, "Draft Proposed Standard for Digital Cartographic Data" *The American Cartographer*, vol. 15, p. 21.

- no headers
- no trailers
- no field padding
- no field separators (at least none required)

in short, no wasted space. With the exception of a defined field separator, these characteristics are also true of meta-data files, which I will call *information files* from here on.

The issue of human readability needs to be discussed. Why is it desirable to have data files be readable by people? Are there penalties to be paid in computer efficiency or in storage usage if human-readable information files are used?

A benefit of human-readable information files is that a manual is not needed to understand something about a data set. It frequently happens that a data set is presented to a potential user without any accompanying documentation. Therefore, understanding something about the data without referring to auxiliary documentation is often useful, and can even be crucial, since you can't read the data unless you know its formats.

Another benefit of human-readable information files is that it is easier to write a parser for these files than if nonprintable data has to be dealt with. Appendix C illustrates parsing of a SPOT[3] data file using the fixed-field data that comes on the SPOT tape. Appendixes B, D, and E illustrate parsing of the same file using a PDEF information file. You can see that the parsing is much more understandable when using PDEF. As illustrated in appendix D, It is also possible to use the parsing tools *lex*[4] and *yacc*,[5] which are of great help in writing parsers. While lex and yacc were created on Unix systems, they are now available under all major operating systems such as MSDOS, OS/2, and VMS.

Human-readable files do take more storage on disk or tape than binary files of the same data. For this reason, PDEF defines two types of files. The human-readable file is only used for data that must be read to understand the data set as a whole. Its inefficient storage is not a problem since binary data files are much larger than human-readable files for the large data sets that PDEF was created to manipulate.

Having mentioned the possible usefulness of PDEF, the PDEF file formats will now be described as will some of the software tools that manipulate them. The body of this report describes the abstract characteristics of PDEF, which are simple, but file formats rely on a consideration of detail. This detail is given in the appendixes which provide examples of possible uses of PDEF.

---

[3] *Format for the SPOT Image Corporation Computer Compatible Tapes*, SPOT Image Corporation, Aug 1986.

[4] M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer generator" in *Unix Time-Sharing System: Unix Progammer's Manual, Vol. 2B 7th edition*, AT&T Bell Laboratories, 1979.

[5] S. C. Johnson, "Yacc: yet another compiler-compiler" in *Unix Time-Sharing System: Unix Progammer's Manual, Vol. 2B 7th edition*, AT&T Bell Laboratories, 1979.

A large data set in PDEF consists of at least two separate files. One of these files is an *information file*, which contains general information about the data such as file offsets and how the data are to be parsed. Appendix B contains a PDEF information file for SPOT satellite data. There is usually only one information file for a given data set. Besides the information file, there is usually at least one *binary data file*. The information file format will be discussed first, followed by a discussion of the binary data file format.

THE INFORMATION FILE

Information files consist only of printable ASCII characters. In the ASCII numeric sequence, printable ASCII characters include characters ' '(space) through '~' (tilde) inclusive and also include the hexadecimal characters 0A (newline) for line breaks and 09 (tab) for spacing. This is all in accordance with the C programming language practice of considering "white space" characters to be among the printable ASCII set, where white space characters are defined as space, tab, and newline. No other characters are allowed in information files.

An information file without nonprintable characters can be easily viewed without bombarding a terminal or workstation with what may be in-band control information, thus putting it into undesirable states. No special programs are needed to view the information. Any program that writes text data onto the screen is usable for viewing information files, and ordinary text manipulation programs can be used to create or modify information files.

Information files have two reserved characters, the pound sign '#' and the colon ':'. The pound sign character '#' indicates a non-parsable comment. Any characters on a line from the first occurrence of a '#' until the end of the line (i.e. until a newline) are not read by the information file parser. The other reserved character is the colon ':' which is used to separate a *key* from the data associated with the key. Appendix A shows an example information file that has been rather strangely formatted to show some of the possible uses of the '#' and ':' characters.

The "key" is purposely not called a "keyword" because a "key" is a phrase that may contain many words. Keys should be designed to be very descriptive. A poor key would be cryptic such as "redoff," while an equivalent good key would be "file offset to beginning of red image." Keys must begin a line. This means that either they must appear at the very beginning of the information file or they must always follow a newline. Keys are separated from the data to which they refer by a colon ':'. Leading and trailing spaces or tabs in a key are ignored by the parser. See the comments in the information file in Appendix A for further details.

Appendix A also shows how a single key may refer to a data structure rather than a single data item. There the structure is a colormap which consists of a repeated sequence of [pixel red green blue] values. Data structures can be defined in a PDEF information file too. For example, there may be entries like this:

```
#
# establish how we encode a colormap
#
colormap sequence: red green blue pixel
color maximum value: 65536 # for the X Window System
pixel maximum value: 255
colormap:
        3456   12345  8976   0
        12345  8974   3458   1
        # and so on ...
```

Other types of composite data, such as matrices or coordinate tuples, may be handled similarly. Here is a possible representation of Quam's block storage of raster data,[6] such as is used for ARC Digitized Raster Graphics, a product of the U.S. Defense Mapping Agency.

```
Data type: raster            # also vector, quadtree, etc.
Storage format: blocked           # could be RGB, band interleaved, etc.
Block size: 128
        # and so on ...
```

An example of how quadtrees may be encoded will be shown later. Appendix F presents a design for storing vector and polygon data.

BINARY DATA FILES

It may be that a data set encoded in PDEF does not contain any binary data files whatever; all of the data being placed in the information file. This is only reasonable, though, for small data sets. For large data sets, one or more binary data files should be used in addition to the information file.

Binary data files contain data in which the largest guaranteed unit of reference is the 8-bit byte. The information file tells the user how to interpret these data bytes. Because of the vagaries of byte ordering on different computers, the information file may specify how to assemble larger data units from bytes. For example, data bytes may be read in the order 1 2 3 4, but a 32-bit integer formed from these bytes may have to be written in the order 2 1 4 3.

Data ordering is a significant problem. The author uses the conventions described in Sun Microsystem's eXternal Data Representation (XDR) standard for representing more complex data types. XDR is explained in the document RFC1014 which may be gotten through the Internet by ftp from nic.sri.com or by request from Sun Microsystems, Inc. Data type encoding is, however, not enforced by PDEF and so will not be discussed here further.

[6] L.H. Quam, "A Storage Representation for Efficient Access to Large Multidimensional Arrays", *Proceedings DARPA Image Understanding Workshop*, 104-111, 1980.

Appendix E shows that a separate information file may be used to work with some data format *without* reformatting it into an intermediate binary format. The programs in Appendix E parse a SPOT file in its distributed format; headers in the SPOT data are just ignored. This technique allows a common set of parsing and data manipulation programs to work on various types of data without reformatting the data file itself. This is a significant advantage for large data sets where reforr atting the data would take much computation and use a large amount of storage.

USE OF PDEF FOR DATA REFORMATTING

As mentioned in the introduction, nothing except valid data should be stored in binary data files. If applicable, one may, of course, specify fixed-length fields, padding, headers, and all the other apparatus found in various file structures. This flexibility has a use in that it is by this means that data can be exchanged from one format to another. It is this problem, data reformatting and exchange, that inspired work on PDEF (and is the source of the name *protean*).

The combinatorics of data reformatting mandate a common intermediate format. The following table shows how the number of parsing programs needed is affected by the presence of an intermediate file format.

Table 1. Number of Parsing Programs Needed

number of formats	parsing programs needed without PDEF	parsing programs needed with PDEF
8	56	16
9	72	18
10	90	20
100	9900	200

The relevant mathematics are that without a common file format one needs $n(n - 1)$ reformatting programs and with a common file format only $2n$ reformatting programs. Thus, the first problem is of order n squared while the second problem is of order n.

The table doesn't tell the whole story, though. The programs that have to be written fall into two equal-sized groups, which is where the factor of 2 in 2n comes from. Figure 1 shows the situation. The programs that convert some other format into PDEF have a common back end in that they all feed into PDEF. Similarly, the programs that convert from PDEF to another format have a common front end that reads the PDEF file. When these commonalities are considered, the problem simplifies further to essentially n programs instead of 2n programs. There is no doubt that an intermediate file format is needed if data reformatting among many formats needs to be done.

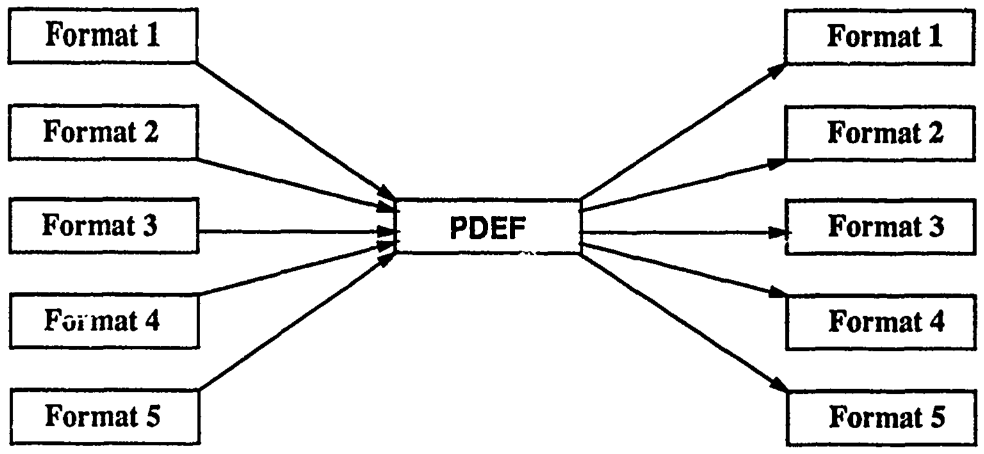

Figure 1: Diagram of interchange among five data formats using PDEF as an intermediate format. All data formats may be reformatted to PDEF and then PDEF may be reformatted into any other format. The arrows represent programs that perform data reformatting and the rectangles represent the formats.

PDEF AS A PRIMARY DATA FORMAT

Besides serving as a means of reformatting data, PDEF can itself be a primary data format. This means that devices such as image scanners and telemetry systems can encode their data in PDEF for transmission. The readability of PDEF information files then allows the operator to quickly check data content. Software systems such as a GIS can also be based on PDEF, thereby allowing easier conversion of data to and from foreign formats. The U. S. Army Engineer Topographic Laboratories (ETL) will use PDEF as the format for data generated by the Terrain Information Extraction System (TIES), which is a developmental system allowing Army units to extract terrain data from digital photography in the field.

TYPES OF DATA THAT MAY BE HANDLED

Current experience with PDEF has been only with regard to raster data files. In order to be useful as a general data exchange file (i.e. in order to be truly *protean*), PDEF has to be able to handle other data types and structures as well. An example relevant to ETL is GIS data, which includes vectors (with associated attributes) and quadtrees as well as gridded (raster) data. Below is an example of a quadtree structure defined in PDEF. Although the following is not the only way to encode a quadtree in PDEF, it is an example meant to indicate that PDEF is capable of doing this.

One way to store a quadtree on a disk or tape is to define a traversal order of the tree and then to linearize the tree to a file by traversing it. To rebuild the tree from the file, just build it in the same order when the file is read. Here is a section of an information file dealing with quadtrees:

```
Traversal order:                              preorder
Quadtree node data order:                     attribute NW NE SE SW
Attribute:                                    generic pointer
NW:                                           boolean
NE:                                           boolean
SE:                                           boolean
SW:                                           boolean
#
# binary data file is just quadtrees, so offset is zero
#
Offset of root quadtree in data file:         0
```

Given this information, a parser program can go through the binary data file and add in the nodes for which the boolean attributes of its father node are TRUE, meaning that there is a node under this quadrant. This simple scheme defines leaf nodes as having all four quadrants FALSE and with an attribute assigned to its area. Note that storage can be saved by encoding the four leaf nodes in a single byte since they only need one bit each to perform their function. Data descriptions, such as "boolean" or "generic pointer," can be further described by other entries in the information file. Other needed data would certainly include geographic coordinates of the corners of the region encoded in the quadtree. Individual quadtree nodes need not carry their coordinates along with them since these are implicit in the tree.

Another data type of great interest to the GIS community is vector data. A realistic design of a vector data format for PDEF is too big to be given in the body of this report; however, Appendix F contains an example design based on a United States Geological Survey format for vector and polygon data, such as is used in a GIS. All essential information such as vector ordering is preserved.

GENERAL DISCUSSION; SYNTAX AND SEMANTICS

PDEF defines the syntax of a data exchange file format and not the semantics of such a format. PDEF does not address some of the most difficult problems associated with reformatting data, such as forcing a match between data fields that are not strictly conformable. If a name field in one format has 30 characters allocated to it and another only allows 10 characters, how is a conversion to be made? Similarly, if needed data in some format is simply not available in a precursor format, what defaults should be used to fill in the blanks in the output format? Should they be filled at all? Problems such as these are matters of policy and are therefore beyond the scope of PDEF because it is only a file formatting vehicle. However, PDEF makes it easier to address these issues of reformatting policy.

For one thing, having a protean and human-readable format for information storage and exchange means that those people charged with deciding the form of data storage or interchange can encode their decisions directly in the information file that is to be parsed by a computer. This makes the data formatting readily and directly reviewable. There is no danger of a mistranslation between what the computer must read and what people can read since information files can be read by both people and computers. Using the information file for this purpose prevents such mismatches. It is best to avoid the production of auxiliary documents detailing formatting decisions since the PDEF information file and the auxiliary document may disagree.

There are some concerns that need to be addressed which come from having a separate file that includes the parsing information for binary data files. These concerns are as follows:

- The information file may be separated from the data files to which it refers.
- The information file is both too easy and too difficult to change. It is too easy to change because it is a text file and can therefore be modified by a text editor so that it no longer has accurate data. It is too difficult to change because a program may modify the binary data without modifying the information file.

In general, a concern exists that there is too much decoupling between the information file and the data it describes.

Although the information file may indeed be separated from the file it describes, many current data formats make use of separate files and this does not seem to be a significant problem. SPOT and ADRG data are both distributed as sets of files for a single coverage area. These files have a complex interrelationship that is much more difficult to work with than the simple scheme described here. Since no documentation exists which relates operational problems caused by these sets of files becoming dissociated, this is probably more of a theoretical than a practical problem.

Having the information file track the data file is a matter of convention that is not enforced by the format itself. A reasonable convention is to have information files be protected so that they are read-only for users and can only be operated on by privileged programs. Data files can be treated in the same way. It is then the responsibility of programs that modify the data to concurrently modify the information file. To be even more certain that information files and data files are consistent, a given set of information and data files should not be modifiable at all except under regulated circumstances. What is meant here is that if the data set is to be modified, it must be copied to a new data set and a new information file generated to describe it. This keeps a history of data processing information, which it is probably desirable to have anyway. Advances in data storage media have alleviated the problem of storage of large data sets, and data can in any case be overwritten, if this is needed.

Pascoe and Penny have recently critiqued the problem of producing a standard interchange format for GIS data.[7] Since they do not start from the assumption that there can be a separation of syntax from semantics for a data exchange format, they are led to the conclusion, repeatedly stated, that any such standard must be very complex. Indeed, if all the decisions about data semantics are considered, then the result is very complex. This report proposes that the format of interchange may itself be very simple and can help with the more difficult policy problems concerning data reformatting.

Data for an output record in one format may have to be derived from a number of files in some other format. This means that searching of several files must be done for these cases to generate a single output datum. Pascoe and Penny advocate reading all input data into a relational database management system (RDBMS) to distribute the data into a set of relations that can then be searched, as needed, for outputting a new format.[8] While this insulates the programmer from explicit searching for data, the redistribution of data may not be a frequent occurrence, and the apparatus of an RDBMS seems unnecessary. This is certainly true of raster data, which will be closely associated in any format and therefore does not have to be redistributed. Pascoe and Penny mention the large amount of computer resources needed when using an RDBMS;[9] file searching as needed should not impose as much of a burden. Experience with PDEF will show whether it is an efficient means of transferring information or whether an RDBMS is a needed adjunct. As mentioned above, much of the initial data may be retained in the original format and only parsed out as needed. The goal should be to do as little reformatting as possible.

SUMMARY AND CONCLUSIONS

PDEF provides a powerful, simple, and human-readable method of encoding data in a form that is easily parsable by automatic computer methods. Since parsers are written in *yacc*, they are expressed in a formal syntax grammar (Backus-Naur form) and are easy to write and modify. PDEF is superior in simplicity and power to other formats and allows efficient data storage. Owing to its simplicity, it is unlikely that future versions of PDEF need to be designed. This ensures that there will not be outdated versions of PDEF that must be accommodated in the future. PDEF does not have the disadvantages pointed out by Pascoe and Penny for other data exchange formats and can serve as a much more tractable and protean standard than current formats.

[7] R. T. Pascoe and J. P. Penney, "Construction of Interfaces for the Exchange of Geographic Data" *Int. J. Geographical Information Systems*, vol. 4, No. 2, 147-156, 1990.

[8] Ibid.

[9] Ibid.

REFERENCES

Information Processing - Specification for a data descriptive file for information interchange, International Organization for Standardization publication ISO 8211-1985(E), 15 Dec 1985.

Digital Cartographic Data Standards Task Force, "Draft Proposed Standard for Digital Cartographic Data" *The American Cartographer*, vol. 15, p. 21.

Format for the SPOT Image Corporation Computer Compatible Tapes, SPOT Image Corporation, Aug 1986.

M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer generator" in *Unix Time-Sharing System: Unix Progammer's Manual, Vol. 2B 7th edition*, AT&T Bell Laboratories, 1979.

S. C. Johnson, "Yacc: yet another compiler-compiler" in *Unix Time-Sharing System: Unix Progammer's Manual, Vol. 2B 7th edition*, AT&T Bell Laboratories, 1979.

L.H. Quam, "A Storage Representation for Efficient Access to Large Multidimensional Arrays", *Proceedings DARPA Image Understanding Workshop*, 104-111, 1980.

R. T. Pascoe and J. P. Penney, "Construction of Interfaces for the Exchange of Geographic Data" *Int. J. Geographical Information Systems*, vol. 4, No. 2, 147-156, 1990.

APPENDIX A. EXAMPLE PDEF INFORMATION FILE.

This appendix contains an illustrative PDEF file which is formatted and commented to show what is possible using PDEF.

```
#
# This is an example of an information file for ETL data such as
# digital images and reformatted DMA data.
# These lines are comment lines and will not be parsed.
#
# Anything after the '#' on a line is ignored; even '#' #####.

# Blank lines are ignored too.

# Keys can appear in either UPPER or lower case or any combination of
# cases as long as they begin at the start of a line and end with a ':'.
# The colon is not part of the key but it is a reserved character
# and may not appear in text.  The only two reserved characters are
# ':' and '#' for keys (':') and comments ('#').  Note that
# colons ::::: are also okay in comments.

WIDTH: 512
# This comment should not appear in the output when this file is parsed.

Height:
                # This comment should not appear either, but -1004
                # should.  Note how values associated with keys
                # may appear anywhere after them and not necessarily
                # on the same line.  Any combination of spaces, tabs,
                # and newlines may appear between a key and its value.
                # Note, however,that a comment line like this
                # may NOT be between a key and its matching value
                # if the matching value is a string.  Otherwise it's
                # okay; like here where the value is an integer: -1004
-1004

# Here is an invalid key to check error handling
foobar: 1024

Comment:
This is a keyed comment.
I'm extending it over several lines to see if the parser keeps the
newlines where I put them.

Including blank lines.

Latitude:123.4564       # Note no space necessary between key and value.
Longitude:N0354566      # This value is a string; Latitude was a
                        # floating-point number

title:  Mike McDonnell's test information file to check parsers.

BITS_PER_PIXEL: 8

# If key phrases are allowed as well as keywords,
# then the following is legal.
BITS PER PIXEL: 24

TYPE: RGB
Tile Size: 128
```

```
Colormap:
# These comment lines and the formatting are only to help the user read
# the colormap info.  The computer just looks for groups of four
# consecutive numbers divided by whitespace.

#    Pixel        Red      Green       Blue
#    -----        ---      -----       ----
         1        256        256        256
         2        512        512        512
         3        768        768        768
         4       1024       1024       1024
         5       1280       1280       1280
         6       1536       1536       1536
         7       1792       1792       1792
         8       2048       2048       2048
         9       2304       2304       2304
        10       2560       2560       2560

# The groups of numbers can even be interrupted with blank lines and
# comments.  No problem.

        11       2816       2816       2816
        12       3072       3072       3072
        13       3328       3328       3328
        14       3584       3584       3584
        15       3840       3840       3840
        16       4096       4096       4096
        17       4352       4352       4352

# Or the data can be irregularly spaced like below.  If there are not some
# multiple of four in the number of numbers the parse will abort, so be
# sure you have full sets of [pixel red green blue] data.
# An aside here is that the parse does not check for ranges of data, but
# some values of any of these numbers will be nonsense.  It is up to the
# application to check the ranges of parsed-out data.

18 3000 2000
5000 255 65536 65536            65536

end:            # Good idea to terminate an information file this way.
```

APPENDIX B. KEY PHRASE FILES IN PDEF AND C.

This appendix contains both a "keys.h" file needed to parse SPOT data and a PDEF file for the same data. The keys.h file is the repository of all the allowed key phrases for a given data format. The PDEF file contains as many of these key phrases as are needed to work with the data at hand. Note that the keys.h file ends with an empty string so that the parser knows when it has reached the end. Also notice that this parser is case independent, so all phrases in keys.h use only lowercase letters. Case independence is not a requirement of PDEF but is a useful convention that may be enforced by a PDEF parser. Case independence is useful because it allows appropriate capitalization to enhance readability of the information files.

Recently, the parsing of the keys file has been simplified by using the same lex and yacc parsers as for an information file. Now a keys file (no longer a "keys.h" file) looks like an information file without values for the keywords. This avoids subjecting the creator of a keys file from following the C language syntax as for the example in this appendix There was not time to include an example of this new keys file without affecting page counts for publication, but as an example, the sequence

```
char   *keys[] =
{
/*
 * Start with the directory file keywords.
 */
 "leader file type (a=ascii e=ebcdic)",
 "leader file type name",
 "leader file class",
 "leader file class code",
 ...
```

is now replaced by

```
#
# Start with the directory file keywords.
#
leader file type (a=ascii e=ebcdic)
leader file type name
leader file class
leader file class code
 ...
```

Notice that trailing colons are not needed after each key since keys are not followed by values in a key definition file. Instead a newline serves to separate keys.

```
/**
 * This is the include file for parsing a SPOT file once it has
 * been put into PDEF format.  A SPOT distribution contains three
 * types of files:
 * 1. tape volume directory files for each tape.
 * 2. a leader file for the data set
 * 3. imagery files which contain information headers as well as pixel data.
 *
 * There is also a trailer file which I do not use.  See SPOT
 * handbooks for more information about these files.
 *
 * Keywords (actually key phrases) are grouped according to which files
 * they are derived from.
 */

char   *keys[] =
{

/*
 * Start with the directory file keywords.
 */
 "leader file type (a=ascii e=ebcdic)",
 "leader file type name",
 "leader file class",
 "leader file class code",
 "leader file data type",
 "leader file data type code",
 "leader number of records in file",
 "leader length of first record of file",
 "leader length of largest record in file",
 "leader file length type",
 "leader file length type code",
 "leader file starts on volume",
 "leader f  e ends on volume",
 "leader location of first file record in this volume",
 "leader number of records in this volume",
 "leader notes",

 "imagery file type (a=ascii e=ebcdic)",
 "imagery file type name",
 "imagery file class",
 "imagery file class code",
 "imagery file data type",
 "imagery file data type code",
 "imagery number of records in file",
 "imagery length of first record of file",
 "imagery length of largest record in file",
 "imagery file length type",
 "imagery file length type code",
 "imagery file starts on volume",
 "imagery file ends on volume",
 "imagery location of first file record in this volume",
 "imagery number of records in this volume",
 "imagery notes",

 "trailer file type (a=ascii e=ebcdic)",
 "trailer file type name",
 "trailer file class",
"trailer file class code",
"trailer file data type",
"trailer file data type code",
"trailer number of records in file",
"trailer length of first record of file",
"trailer length of largest record in file",
"trailer file length type",
"trailer file length type code",
"trailer file starts on volume",
"trailer file ends on volume",
"trailer location of first file record in this volume",
"trailer number of records in this volume",
"trailer notes",

/*
 * These are the leader file keywords.
 */
"center pixel latitude",
"center pixel longitude",
"center pixel row",
"center pixel column",

"corner 1 pixel latitude",
"corner 1 pixel longitude",
"corner 1 pixel row",
"corner 1 pixel column",

"corner 2 pixel latitude",
"corner 2 pixel longitude",
"corner 2 pixel row",
"corner 2 pixel column",

"corner 3 pixel latitude",
"corner 3 pixel longitude",
"corner 3 pixel row",
"corner 3 pixel column",

"corner 4 pixel latitude",
"corner 4 pixel longitude",
"corner 4 pixel row",
"corner 4 pixel column",

"nadir latitude",
"nadir longitude",

"photo orientation",
"direction of incidence (l=left r=right)",
"photo angle of incidence with ellipsoid at center",

"sun azimuth",
"sun elevation",

"year",
"month",
"day",
"hour",
"minute",
"second",
"mission id",
"sensor id",
"spectral mode",

"pixels per line",
"number of lines",

"band interleave factor",
"number of bands",

"preprocess level",
"recalibration designator",
"deconvolution designator",
"resampling designator",

"pixel width on ground in meters",
"pixel height on ground in meters",

/*
 * These are the imagery file keywords
 */
 "number of image records",
 "image record length",
 "bits per pixel",
 "pixels per pixel group",
 "bytes per pixel group",
 "pixel ordering in group",
 "number of spectral bands",
 "lines per image not including borders",
 "left border pixels per line",
 "image pixels per line",
 "right border pixels per line",
 "number of top border lines",
 "number of bottom border lines",
 "interleave type",
 "data records per line",
 "data records per multispectral line",
 "bytes of prefix data per record",
 "bytes of image data per record",
 "bytes of suffix data per record",
 "prefix/suffix repeat flag",
 "scan line number locator",
 "image (band) number locator",
 "time of scan line locator",
 "left-fill count locator",
 "right-fill count locator",
 "left-fill bits within pixel",
 "right-fill bits within pixel",
 "maximum value of pixel",

/*
 * Added to all keys[] arrays to provide termination.  -mmm
 */
 "end",
 "",                              /* need this to find end of list. */
};
```

```
#
# An example PDEF information file for a SPOT image.
# Data was derived from a 2-tape SPOT dataset of Ft. Hood, TX area.
#

#
# First the volume directory data with a block for each of the
# other file types in the distribution.  There is a leader file,
# an imagery file, and a trailer file in addition to the directory
# file itself.
#
leader File type (A=ASCII E=EBCDIC): A
leader File type name: SP1 X1B LEADBIL
leader File class: LEADER FILE
leader File class code: LEAD
leader File data type: MIXED BINARY AND ASCII
leader File data type code: MBAA
leader Number of records in file: 27
leader Length of first record of file: 3960
leader Length of largest record in file: 3960
leader File length type: FIXED LENGTH
leader File length type code: FIXD
leader File starts on volume: 1
leader File ends on volume: 1
leader Location of first file record in this volume: 1
leader Number of records in this volume: 0
leader Notes:

imagery File type (A=ASCII E=EBCDIC): A
imagery File type name: SP1 X1B IMGYBIL
imagery File class: IMAGERY FILE
imagery File class code: IMGY
imagery File data type: BINARY ONLY
imagery File data type code: BINO
imagery Number of records in file: 9014
imagery Length of first record of file: 5400
imagery Length of largest record in file: 5400
imagery File length type: FIXED LENGTH
imagery File length type code: FIXD
imagery File starts on volume: 1
imagery File ends on volume: 2
imagery Location of first file record in this volume: 1
imagery Number of records in this volume: 4507
imagery Notes:

trailer File type (A=ASCII E=EBCDIC): A
trailer File type name: SP1 X1B TRAIBIL
trailer File class: TRAILER FILE
trailer File class code: TRAI
trailer File data type: MIXED BINARY AND ASCII
trailer File data type code: MBAA
trailer Number of records in file: 3
trailer Length of first record of file: 1080
trailer Length of largest record in file: 1080
trailer File length type: FIXED LENGTH
trailer File length type code: FIXD
trailer File starts on volume: 2
trailer File ends on volume: 2
trailer Location of first file record in this volume: 1
trailer Number of records in this volume: 0
trailer Notes:

#
# Here is the data from the imagery file
#
Number of image records: 9012
Image record length: 5400
Bits per pixel: 8
Pixels per pixel group: 1
Bytes per pixel group: 1
Pixel ordering in group: 0
Number of spectral bands: 3
Lines per image not including borders: 3004
Left border pixels per line: 0
Image pixels per line: 3179
Right border pixels per line: 2121
Number of top border lines: 0
Number of bottom border lines: 0
Interleave type: BIL
Data records per line: 1
Data records per multispectral line: 3
Bytes of prefix data per record: 20
Bytes of image data per record: 5300
Bytes of suffix data per record: 28
Prefix/suffix repeat flag:
line locator:    1 4PB
band locator:    5 4PB
Time locator:
Left locator:   13 4PB
Right locator:   17 4PB
Left-fill bits within pixel: 0
Right-fill bits within pixel: 0
Maximum value of pixel: 254

#
# Here is the data from the leader file
#
Center pixel Latitude: N0310627
Center pixel Longitude: W0974234
Center pixel Row: 1502
Center pixel Column: 1584

Corner 1 pixel Latitude: N0312510
Corner 1 pixel Longitude: W0975633
Corner 1 pixel Row: 1
Corner 1 pixel Column: 179

Corner 2 pixel Latitude: N0311915
Corner 2 pixel Longitude: W0971923
Corner 2 pixel Row: 1
Corner 2 pixel Column: 3179
Corner 3 pixel Latitude: N0305333
Corner 3 pixel Longitude: W0980532
Corner 3 pixel Row: 3004
Corner 3 pixel Column: 1

Corner 4 pixel Latitude: N0304740
Corner 4 pixel Longitude: W0972834
Corner 4 pixel Row: 3004
Corner 4 pixel Column: 2998

Nadir Latitude: N0310652
Nadir Longitude: W0981948

Photo orientation: 10.500000

Direction of incidence (L=left R=right): L
Photo angle of incidence with ellipsoid at center: 4.600000

Sun azimuth: 132.100006
Sun elevation: 69.000000

year: 1988
month: 05
day: 04
hour: 17
minute: 23
second: 05

Mission ID: SPOT1
Sensor ID: HRV1
Spectral mode: XS

Pixels per line: 3179
Number of lines: 3004

Band interleave factor: BIL
Number of bands: 3

Preprocess level: 1B
Recalibration designator: 1
Deconvolution designator: 1
Resampling designator: CC

Pixel width on ground in meters: 20
Pixel height on ground in meters: 20
END:
```

APPENDIX C. C PROGRAM TO PARSE SPOT DATA WITHOUT PDEF.

This appendix contains the C language code that generates a PDEF file from raw SPOT data. This code is rather complex and illustrates the difficulties in parsing a typical data format.

```
/*
 * Struct for fixed part of each record, which is binary numbers. This
 * is always found in the first 12 bytes of each record.  All other
 * information in the directory, leader, and trailer files are printable
 * chars, usually ASCII.
 */

/* A portable integer type */
struct r_int
{
  char    byte1, byte2, byte3, byte4;
};

struct fixed
{
  struct r_int record_num;       /* assumes 4-byte int; little-endian */
  char    sub1;                  /* first record sub-type */
  char    type;                  /* record type */
  char    sub2;                  /* second record sub-type; not used */
  char    sub3;                  /* third record sub-type; not used */
  struct r_int record_length;
};

/*
 * Here is the data necessary to read a SPOT volume directory file.  The
 * volume directory is the first file on any SPOT tape.  Parsing the
 * volume directory then gives you the information to read the other
 * files on the tape.
 *
 * The first record on the directory file is local to the file.  The second
 * record gives information about the leader file, the third record is
 * for the imagery file and the fourth record is for the trailer file.
 *
 * I define two structs.  The first, fp, is used to overlay the
 * fixed-field-length format of the record and the second, cfp,  is used
 * to convert the information in fp to standard C data types such as
 * strings, ints and floats.  The information in cfp can then be used to
 * manipulate the files on the tapes.
 */

#define DIRECTORY_SIZE 1800      /* fixed-size file */
#define DIRECTORY_RECORD_SIZE 360       /* for directory file only */

/*
 * This is the structure for a header file file-pointer record
 */

struct fp
{
  char    fixed[12];             /* binary data describing the file */
  char    type[2];               /* A=ASCII E=EBCDIC */
  char    blank[2];              /* not used */
  char    num[4];                /* 1=leader 2=imagery 3=trailer */
  char    name[16];              /* name of file type (leader;imagery,... */
  char    class[28];
  char    class_code[4];
  char    data_type[28];
  char    data_type_code[4];
 char    num_records[8];
 char    length1[8];           /* length of first record */
 char    maxlength[8];         /* biggest record */
 char    length_type[12];      /* file length type (?) */
 char    length_code[4];
 char    start_volume[2];      /* what volume file starts on */
 char    end_volume[2];
 char    first_record[8];      /* location of first record in this
                                * volume. */
 char    blank2[108];
 char    num_records_vol[8];   /* number of records in this volume */
 char    local_use[92];
};

/*
 * This is the above struct converted to standard C data types for ease
 * of handling.
 */
struct cfp
{
 char    fixed[13];            /* binary data describing the file */
 char    type[3];              /* A=ASCII E=EBCDIC */
 char    name[17];             /* name of file type (leader;imagery,... */
 char    class[29];
 char    class_code[5];
 char    data_type[29];
 char    data_type_code[5];
 int     num_records;
 int     length1;              /* length of first record */
 int     maxlength;            /* biggest record */
 char    length_type[13];      /* file length type */
 char    length_code[4];
 int     start_volume;         /* what volume file starts on */
 int     end_volume;
 int     first_record;         /* where is first record in this volume? */
 int     num_records_vol;      /* number of records in this volume */
 char    local_use[93];
};

/*
 * Leader file-header record.  Not all fields are represented, only the
 * ones I was interested in.
 */

struct lfh
{
 char    fixed[12];            /* binary data describing the file */
 char    junk1[72];            /* stuff I don't care about */
 char    cen_lat[16];          /* latitude of center pixel */
 char    cen_long[16];         /* longitude of center pixel */
 char    cen_row[16];          /* row number of center pixel */
 char    cen_col[16];          /* column number of center pixel */
 char    cor1_lat[16];         /* Upper Left corner data */
 char    cor1_long[16];
 char    cor1_row[16];
 char    cor1_col[16];
 char    cor2_lat[16];         /* Upper Right corner data */
 char    cor2_long[16];
```

```
  char    cor2_row[16];
  char    cor2_col[16];
  char    cor3_lat[16];          /* Lower Left corner data.*/
  char    cor3_long[16];
  char    cor3_row[16];
  char    cor3_col[16];
  char    cor4_lat[16];          /* Lower Right corner data */
  char    cor4_long[16];
  char    cor4_row[16];
  char    cor4_col[16];
  char    nadir_lat[16];         /* Satellite nadir when center pixel
                                  * taken */
  char    nadir_long[16];
  char    orientation[16];       /* Image line rotation from East (+ CCW) */
  char    incidence[16];         /* angle of optical axis to ellipsoid */
  char    sun_az[16];            /* Sun location */
  char    sun_elev[16];
  char    junk2[80];
  char    center_time[32];       /* time of taking center pixel */
  char    mission_id[16];        /* see SPOT manual for these next 3 */
  char    sensor_id[16];
  char    spectral_mode[16];
  char    junk3[336];
  char    pixels_per_line[16];   /* this includes line header and trailer */
  char    num_lines[16];         /* including header line, which has no
                                  * data */
  char    interleave[16];        /* BIL if band interleave format */
  char    num_bands[16];         /* number of spectral bands present */
  char    band_indicator[256];   /* ?? */
  char    preprocess[16];        /* see SPOT manual for these next 4
                                  * members */
  char    recalib_designator[16];
  char    deconvolve_designator[16];
  char    resample_designator[16];
  char    pixel_width[16];       /* in arc seconds */
  char    pixel_height[16];
  /* There are lots of remaining fields, but I got the ones I wanted. */
};

/*
 * Converted Leader file-header record.  Not all fields are represented,
 * only the ones I was interested in.  See the comments on the
 * corresponding fields above.
 */

struct clfh
{
  char    fixed[13];             /* binary data describing the file */
  char    cen_lat[17];
  char    cen_long[17];
  int     cen_row;
  int     cen_col;
  char    cor1_lat[17];
  char    cor1_long[17];
  int     cor1_row;
  int     cor1_col;
  char    cor2_lat[17];
  char    cor2_long[17];
 int     cor2_row;
 int     cor2_col;
 char    cor3_lat[17];
 char    cor3_long[17];
 int     cor3_row;
 int     cor3_col;
 char    cor4_lat[17];
 char    cor4_long[17];
 int     cor4_row;
 int     cor4_col;
 char    nadir_lat[17];
 char    nadir_long[17];
 float   orientation;
 char    incidence_dir;
 float   incidence;
 float   sun_az;
 float   sun_elev;
 char    center_time_yr[5];
 char    center_time_mo[3];
 char    center_time_da[3];
 char    center_time_hr[3];
 char    center_time_mn[3];
 char    center_time_sc[3];
 char    mission_id[17];
 char    sensor_id[17];
 char    spectral_mode[17];
 int     pixels_per_line;
 int     num_lines;
 char    interleave[17];
 int     num_bands;
 char    band_indicator[257];  /* ?? */
 char    preprocess[17];
 char    recalib_designator[17];
 char    deconvolve_designator[17];
 char    resample_designator[17];
 int     pixel_width;
 int     pixel_height;
 /* There are lots of remaining fields, but I got the ones I wanted. */
};

/*
 * This is the include file to parse SPOT imagery files The imagery file
 * mostly contains pixels, but the file has a header (the first record)
 * that gives format information needed to read the file.  Struct im is
 * in the SPOT fixed format and is used to overlay the first record in
 * the imagery file(s) to get the data needed.  The second struct, cim,
 * is used to store the parsed data as standard C types and to convert
 * data to a PDEF format.
 */

struct im
{
  char    fixed[180];          /* binary data describing the file */
  char    num_records[6];      /* Number of image records in file */
  char    record_length[6];    /* See SPOT manual for allowed lengths */
  char    junk1[24];           /* not used */
  /* Pixel group data */
  char    bits_per_pixel[4];
  char    pixels_per_group[4];  /* What is a data group? */
  char    bytes_per_group[4];
  char    order_in_group[4];

  /* Image data */
  char    num_bands[4];         /* number of spectral bands */
  char    num_lines[8];         /* excluding top, bottom, border lines */
  char    num_left[4];          /* number of left border lines */
  char    num_image_pixels[8];  /* per line, including borders */
  char    num_right[4];
  char    num_top[4];           /* number of top border lines */
  char    num_bottom[4];
  char    interleave[4];        /* BIL, etc.  See manual for types */

  /* Record data for this file */
  char    recs_per_line[2];     /* usually 1 */
  char    recs_per_ms_line[2];  /* multispectral line, usually 3 records */
  char    prefix[4];            /* prefix bytes per line, always 32 */
  char    image[8];             /* image data bytes per line, incl.
                                 * borders */
  char    suffix[4];            /* suffix bytes per line, always 68 */
  char    repeat[4];            /* prefix, suffix repeat flag (?) */

  /* Prefix and suffix data locators */
  char    line[8];
  char    band[8];
  char    time[8];
  char    left_fill[8];
  char    right_fill[8];
  char    junk2[96];            /* not interested for now */

  /* Per-pixel data description */
  char    left_fill_bits[4];
  char    right_fill_bits[4];
  char    max_range[8];
};

/*
 * This struct contains the C data types corresponding to the header
 * fields picked out by struct im just above.
 */
struct cim
{
  char    fixed[181];           /* fixed data that starts out */
  int     num_records;
  int     record_length;
  int     bits_per_pixel;
  int     pixels_per_group;
  int     bytes_per_group;
  int     order_in_group;
  int     num_bands;
  int     num_lines;
  int     num_left;
  int     num_image_pixels;
  int     num_right;
  int     num_top;
  int     num_bottom;
  char    interleave[5];
  int     recs_per_line;
  int     recs_per_ms_line;
  int     prefix;
  int     image;
  int     suffix;
  char    repeat[5];
  char    line[9];
  char    band[9];
  char    time[9];
  char    left_fill[9];
  char    right_fill[9];
  int     left_fill_bits;
  int     right_fill_bits;
  int     max_range;
};

/*
 * This function takes as input a buffer containing the directory record
 * of the volume directory file for a spot image and it parses the
 * buffer to change the data into C entities such as null-teminated
 * strings, ints and floats. The data are returned in a cfp struct which
 * is defined in volume.h.
 */

#include <string.h>
#include "volume.h"


int
read_directory(cpp, buffer)
  struct cfp *cpp;
  char   *buffer;
{
  struct fp *hp;
  extern float tofloat();

  hp = (struct fp *) (buffer);

  strncpy(cpp->fixed, hp->fixed, 12);

  strncpy(cpp->type, hp->type, 2);
  strncpy(cpp->name, hp->name, 16);
  strncpy(cpp->class, hp->class, 28);
  strncpy(cpp->class_code, hp->class_code, 4);
  strncpy(cpp->data_type, hp->data_type, 28);
  strncpy(cpp->data_type_code, hp->data_type_code, 4);
  cpp->num_records = toint(hp->num_records, 8);
  cpp->lenghtl = toint(hp->lengthl, 8);
  cpp->maxlength = toint(hp->maxlength, 8);

  strncpy(cpp->length_type, hp->length_type, 12);
  strncpy(cpp->length_code, hp->length_code, 4);

  cpp->start_volume = toint(hp->start_volume, 2);
  cpp->end_volume = toint(hp->end_volume, 2);
  cpp->first_record = toint(hp->first_record, 8);
  cpp->num_records_vol = toint(hp->num_records_vol, 8);
  strncpy(cpp->local_use, hp->local_use, 92);
}
#include <stdio.h>
#include "volume.h"

print_directory(cpp, prefix)
  struct cfp *cpp;
  char       *prefix;
{
  printf("%s ", prefix);
  printf("File type (A=ASCII E=EBCDIC): %s\n", cpp->type);
  printf("%s ", prefix);
  printf("File type name: %s\n", cpp->name);
  printf("%s ", prefix);
  printf("File class: %s\n", cpp->class);
  printf("%s ", prefix);
  printf("File class code: %s\n", cpp->class_code);
  printf("%s ", prefix);
  printf("File data type: %s\n", cpp->data_type);
  printf("%s ", prefix);
  printf("File data type code: %s\n", cpp->data_type_code);
  printf("%s ", prefix);
  printf("Number of records in file: %d\n", cpp->num_records);
  printf("%s ", prefix);
  printf("Length of first record of file: %d\n", cpp->length1);
  printf("%s ", prefix);
  printf("Length of largest record in file: %d\n", cpp->maxlength);
  printf("%s ", prefix);
  printf("File length type: %s\n", cpp->length_type);
  printf("%s ", prefix);
  printf("File length type code: %s\n", cpp->length_code);
  printf("%s ", prefix);
  printf("File starts on volume: %d\n", cpp->start_volume);
  printf("%s ", prefix);
  printf("File ends on volume: %d\n", cpp->end_volume);
  printf("%s ", prefix);
  printf("Location of first file record in this volume: %d\n",
         cpp->first_record);
  printf("%s ", prefix);
  printf("Number of records in this volume: %d\n", cpp->num_records_vol);
  printf("%s ", prefix);
  printf("Notes: %s\n", cpp->local_use);

  putchar('\n');
}

/*
 * This function takes as input a buffer containing the directory record
 * of the leader file for a spot image and it parses the buffer to
 * change the data into C entities such as null-teminated strings, ints
 * and floats. The data are returned in a clfh struct which is defined
 * in leader.h.
 */

#include <string.h>
#include "leader.h"

#define RECORD_SIZE 3960        /* from parsing directory file */
```

```
int
read_leader(cpp, buffer)
  struct clfh *cpp;
  char   *buffer;
{
  struct lfh *hp;
  extern float tofloat();

  hp = (struct lfh *) (buffer + RECORD_SIZE);

  strncpy(cpp->fixed, hp->fixed, 12);

  strncpy(cpp->cen_lat, hp->cen_lat, 16);
  strncpy(cpp->cen_long, hp->cen_long, 16);
  cpp->cen_row = toint(hp->cen_row, 16);
  cpp->cen_col = toint(hp->cen_col, 16);

  strncpy(cpp->cor1_lat, hp->cor1_lat, 16);
  strncpy(cpp->cor1_long, hp->cor1_long, 16);
  cpp->cor1_row = toint(hp->cor1_row, 16);
  cpp->cor1_col = toint(hp->cor1_col, 16);

  strncpy(cpp->cor2_lat, hp->cor2_lat, 16);
  strncpy(cpp->cor2_long, hp->cor2_long, 16);
  cpp->cor2_row = toint(hp->cor2_row, 16);
  cpp->cor2_col = toint(hp->cor2_col, 16);

  strncpy(cpp->cor3_lat, hp->cor3_lat, 16);
  strncpy(cpp->cor3_long, hp->cor3_long, 16);
  cpp->cor3_row = toint(hp->cor3_row, 16);
  cpp->cor3_col = toint(hp->cor3_col, 16);

  strncpy(cpp->cor4_lat, hp->cor4_lat, 16);
  strncpy(cpp->cor4_long, hp->cor4_long, 16);
  cpp->cor4_row = toint(hp->cor4_row, 16);
  cpp->cor4_col = toint(hp->cor4_col, 16);

  strncpy(cpp->nadir_lat, hp->nadir_lat, 16);
  strncpy(cpp->nadir_long, hp->nadir_long, 16);

  cpp->orientation = tofloat(hp->orientation, 16);

  cpp->incidence_dir = *hp->incidence;  /* first char incidence is
                                         * direction */
  cpp->incidence = tofloat(hp->incidence + 1, 15);
  cpp->sun_az = tofloat(hp->sun_az, 16);
  cpp->sun_elev = tofloat(hp->sun_elev, 16);

  strncpy(cpp->center_time_yr, hp->center_time, 4);
  strncpy(cpp->center_time_mo, hp->center_time + 4, 2);
  strncpy(cpp->center_time_da, hp->center_time + 6, 2);
  strncpy(cpp->center_time_hr, hp->center_time + 8, 2);
  strncpy(cpp->center_time_mn, hp->center_time + 10, 2);
  strncpy(cpp->center_time_sc, hp->center_time + 12, 2);

  strncpy(cpp->mission_id, hp->mission_id, 16);
  strncpy(cpp->sensor_id, hp->sensor_id, 16);
  strncpy(cpp->spectral_mode, hp->spectral_mode, 16);
  cpp->pixels_per_line = toint(hp->pixels_per_line, 16);
  cpp->num_lines = toint(hp->num_lines, 16);

  strncpy(cpp->interleave, hp->interleave, 16);
  cpp->num_bands = toint(hp->num_bands, 16);
  strncpy(cpp->band_indicator, hp->band_indicator, 256);
  strncpy(cpp->preprocess, hp->preprocess, 16);
  strncpy(cpp->recalib_designator, hp->recalib_designator, 16);
  strncpy(cpp->deconvolve_designator, hp->deconvolve_designator, 16);
  strncpy(cpp->resample_designator, hp->resample_designator, 16);
  cpp->pixel_width = toint(hp->pixel_width, 16);
  cpp->pixel_height = toint(hp->pixel_height, 16);
}

#include <stdio.h>
#include "leader.h"

int
print_leader(cpp)
  struct clfh *cpp;
{
  printf("\nCenter pixel Latitude: %s", cpp->cen_lat);
  printf("\nCenter pixel Longitude: %s", cpp->cen_long);
  printf("\nCenter pixel Row: %d", cpp->cen_row);
  printf("\nCenter pixel Column: %d", cpp->cen_col);

  printf("\n\nCorner 1 pixel Latitude: %s", cpp->cor1_lat);
  printf("\nCorner 1 pixel Longitude: %s", cpp->cor1_long);
  printf("\nCorner 1 pixel Row: %d", cpp->cor1_row);
  printf("\nCorner 1 pixel Column: %d", cpp->cor1_col);

  printf("\n\nCorner 2 pixel Latitude: %s", cpp->cor2_lat);
  printf("\nCorner 2 pixel Longitude: %s", cpp->cor2_long);
  printf("\nCorner 2 pixel Row: %d", cpp->cor2_row);
  printf("\nCorner 2 pixel Column: %d", cpp->cor2_col);

  printf("\n\nCorner 3 pixel Latitude: %s", cpp->cor3_lat);
  printf("\nCorner 3 pixel Longitude: %s", cpp->cor3_long);
  printf("\nCorner 3 pixel Row: %d", cpp->cor3_row);
  printf("\nCorner 3 pixel Column: %d", cpp->cor3_col);

  printf("\n\nCorner 4 pixel Latitude: %s", cpp->cor4_lat);
  printf("\nCorner 4 pixel Longitude: %s", cpp->cor4_long);
  printf("\nCorner 4 pixel Row: %d", cpp->cor4_row);
  printf("\nCorner 4 pixel Column: %d", cpp->cor4_col);
  printf("\n\nNadir Latitude: %s", cpp->nadir_lat);
  printf("\nNadir Longitude: %s", cpp->nadir_long);

  printf("\n\nPhoto orientation: %f", cpp->orientation);

  printf("\n\nDirection of incidence (L=left R=right): %c",
         cpp->incidence_dir);
  printf("\nPhoto angle of incidence with ellipsoid at center: %f",
         cpp->incidence);

  printf("\n\nSun azimuth: %f", cpp->sun_az);
  printf("\nSun elevation: %f", cpp->sun_elev);
  printf("\n\nYear: %s\nMonth: %s\nDay: %s\nHour: %s\nMinute: %s\nSecond: %s",
         cpp->center_time_yr, cpp->center_time_mo, cpp->center_time_da,
         cpp->center_time_hr, cpp->center_time_mn, cpp->center_time_sc);
  printf("\n\nMission ID: %s", cpp->mission_id);
  printf("\nSensor ID: %s", cpp->sensor_id);
  printf("\nSpectral mode: %s", cpp->spectral_mode);

  printf("\n\nPixels per line: %d", cpp->pixels_per_line);
  printf("\nNumber of lines: %d", cpp->num_lines);

  printf("\n\nBand interleave factor: %s", cpp->interleave);
  printf("\nNumber of bands: %d", cpp->num_bands);

  printf("\n\nPreprocess level: %s", cpp->preprocess);
  printf("\nRecalibration designator: %s", cpp->recalib_designator);
  printf("\nDeconvolution designator: %s", cpp->deconvolve_designator);
  printf("\nResampling designator: %s", cpp->resample_designator);

  printf("\n\nPixel width on ground in meters: %d", cpp->pixel_width);
  printf("\nPixel height on ground in meters: %d", cpp->pixel_height);

  putchar('\n');
}

/*
 * This function takes as input a buffer containing the directory record
 * of the imagery file for a spot image and it parses the buffer to
 * change the data into C entities such as null-teminated strings, ints
 * and floats. The data are returned in a cim struct which is defined in
 * imagery.h.
 */

#include <string.h>
#include "imagery.h"

int
read_imagery(cpp, buffer)
  struct cim *cpp;
  char   *buffer;
{
  struct im *hp;

  hp = (struct im *) (buffer);

  strncpy(cpp->fixed, hp->fixed, 180);
  cpp->num_records = toint(hp->num_records, 6);
  cpp->record_length = toint(hp->record_length, 6);

  /* Pixel group data */
  cpp->bits_per_pixel = toint(hp->bits_per_pixel, 4);
  cpp->pixels_per_group = toint(hp->pixels_per_group, 4);
  cpp->bytes_per_group = toint(hp->bytes_per_group, 4);
  cpp->order_in_group = toint(hp->order_in_group, 4);

  /* Image data */
  cpp->num_bands = toint(hp->num_bands, 4);
  cpp->num_lines = toint(hp->num_lines, 8);
  cpp->num_left = toint(hp->num_left, 4);
  cpp->num_image_pixels = toint(hp->num_image_pixels, 8);
  cpp->num_right = toint(hp->num_right, 4);
  cpp->num_top = toint(hp->num_top, 4);
  cpp->num_bottom = toint(hp->num_bottom, 4);
  strncpy(cpp->interleave, hp->interleave, 3);

  /* Record data in this imagery file */
  cpp->recs_per_line = toint(hp->recs_per_line, 2);
  cpp->recs_per_ms_line = toint(hp->recs_per_ms_line, 2);
  cpp->prefix = toint(hp->prefix, 4);
  cpp->image = toint(hp->image, 8);
  cpp->suffix = toint(hp->suffix, 4);
  strncpy(cpp->repeat, hp->repeat, 4);

  /* Prefix and suffix data locators */
  strncpy(cpp->line, hp->line, 8);
  strncpy(cpp->band, hp->band, 8);
  strncpy(cpp->time, hp->time, 8);
  strncpy(cpp->left_fill, hp->left_fill, 8);
  strncpy(cpp->right_fill, hp->right_fill, 8);

  /* Pixel data description */
  cpp->left_fill_bits = toint(hp->left_fill_bits, 4);
  cpp->right_fill_bits = toint(hp->right_fill_bits, 4);
  cpp->max_range = toint(hp->max_range, 8);

}
#include <stdio.h>
#include "imagery.h"

#define RECORD_SIZE 5400

print_imagery(cpp)
  struct cim *cpp;
{
  putchar('\n');

  printf("\nNumber of image records: %d", cpp->num_records);
  printf("\nImage record length: %d", cpp->record_length);
  printf("\nBits per pixel: %d", cpp->bits_per_pixel);
  printf("\nPixels per pixel group: %d", cpp->pixels_per_group);
  printf("\nBytes per pixel group: %d", cpp->bytes_per_group);
  printf("\nPixel ordering in group: %d", cpp->order_in_group);
  printf("\nNumber of spectral bands: %d", cpp->num_bands);
  printf("\nLines per image not including borders: %d",
         cpp->num_lines);
  printf("\nLeft border pixels per line: %d", cpp->num_left);
  printf("\nImage pixels per line: %d", cpp->num_image_pixels);
  printf("\nRight border pixels per line: %d", cpp->num_right);
  printf("\nNumber of top border lines: %d", cpp->num_top);
  printf("\nNumber of bottom border lines: %d", cpp->num_bottom);
  printf("\nInterleave type: %s", cpp->interleave);

  printf("\nData records per line: %d", cpp->recs_per_line);
  printf("\nData records per multispectral line: %d",
         cpp->recs_per_ms_line);
  printf("\nBytes of prefix data per record: %d", cpp->prefix);
  printf("\nBytes of image data per record: %d", cpp->image);
  printf("\nBytes of suffix data per record: %d", cpp->suffix);
  printf("\nPrefix/suffix repeat flag: %s", cpp->repeat);

  printf("\nScan line number locator: %s", cpp->line):
  printf("\nImage (band) number locator: %s", cpp->band);
  printf("\nTime of scan line locator: %s", cpp->time);
  printf("\nLeft-fill count locator: %s", cpp->left_fill);
  printf("\nRight-fill count locator: %s", cpp->right_fill);

  printf("\nLeft-fill bits within pixel: %d", cpp->left_fill_bits);
  printf("\nRight-fill bits within pixel: %d", cpp->right_fill_bits);
  printf("\nMaximum value of pixel: %d", cpp->max_range);

  putchar('\n');
}

/*
 * This program reads SPOT data in and writes out separate arrays for
 * the green, red, and infrared bands.  This only works for multiband
 * data in band interleaved format.
 */

#include <stdio.h>
#include <sys/file.h>
#include "volume.h"
#include "imagery.h"

main(argc, argv)
  int     argc;
  char   **argv;
{
  char    *buffer;
  char    *green, *red, *infrared;        /* unpacked image line buffers */
  int     i;
  int     fd,                  /* general-purpose file descriptor */
          fdg, fdr, fdir,      /* file descriptors for unpacked data */
          line_length,         /* number of bytes per imagery record */
          line_front,          /* number of junk pixels at line start */
          line_back;           /* number of junk pixels at line end */
  int     image_width;         /* number of image pixels per line */
  struct cfp ds1;              /* directory struct for first volume */
  struct cfp ds2;              /* directory struct for second volume */
  struct cim is;               /* imagery header struct */
  extern int toint(), read_leader();
  extern char *malloc();
  extern char *realloc();

  if (argc < 4)
    fprintf(stderr, "Usage: %s directory1 directory2 imagery1 imagery2\n",
            argv[0]), exit(0);

  /* Open and read volume directories to get info about imagery files */
  buffer = malloc(DIRECTORY_SIZE);
  if ((fd = open(argv[1], O_RDONLY, 0775)) <= 0)
    perror("open volume directory failed"), exit(1);
  if (read(fd, buffer, DIRECTORY_SIZE) != DIRECTORY_SIZE)
    perror("read volume directory failed"), exit(1);
  /* imagery info is in the third record */
  read_directory(&ds1, buffer + 2 * DIRECTORY_RECORD_SIZE);
  close(fd);

  if ((fd = open(argv[2], O_RDONLY, 0775)) <= 0)
    perror("open volume directory failed"), exit(1);
  if (read(fd, buffer, DIRECTORY_SIZE) != DIRECTORY_SIZE)
    perror("read volume directory failed"), exit(1);
  read_directory(&ds2, buffer + 2 * DIRECTORY_RECORD_SIZE);
  close(fd);

  /* Open and read the first record of the first imagery file */
  line_length = ds1.maxlength;
  buffer = realloc(buffer, line_length);
  if ((fd = open(argv[3], O_RDONLY, 0775)) <= 0)
    perror("open imagery failed"), exit(1);
  if (read(fd, buffer, line_length) != line_length)
    perror("read imagery failed"), exit(1);
  read_imagery(&is, buffer);

  line_front = 32 + is.num_left;
  line_back = is.num_right + 68;
  image_width = is.num_image_pixels;

  /*
   * Imagery is in two files; necessary info is in two volume directory
   * files and in the imagery header.  The leader file is not important
   * at this stage.
   */
  /* Open files to receive the unpacked image as three color arrays */
  if ((fdg = open("green", O_WRONLY | O_CREAT, 0775)) <= 0)
    perror("open green failed"), exit(1);
  if ((fdr = open("red", O_WRONLY | O_CREAT, 0775)) <= 0)
    perror("open red failed"), exit(1);
  if ((fdir = open("infrared", O_WRONLY | O_CREAT, 0775)) <= 0)
    perror("open infrared failed"), exit(1);

  green = malloc(image_width);
  red = malloc(image_width);
  infrared = malloc(image_width);
#ifdef FULL_IMAGE

  /*
   * Read in the first half of the image from the first imagery file. We
   * are currently at the beginning of the first image data line.
   */
  for (i = 0; i < (ds1.num_records_vol - 1) / 3; ++i)
  {
    lseek(fd, (long) line_front, 1);
    read(fd, green, image_width);
    write(fdg, green, image_width);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, red, image_width);
    write(fdr, red, image_width);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, infrared, image_width);
    write(fdir, infrared, image_width);
    lseek(fd, (long) line_back, 1);
  }
```

```
 /* Open and read the first record of the second imagery file */
 line_length = ds2.maxlength;
 buffer = realloc(buffer, line_length);
 if ((fd = open(argv[4], O_RDONLY)) == 0)
   perror("open imagery failed"), exit(1);
 if (read(fd, buffer, line_length) != line_length)
   perror("read imagery failed"), exit(1);
 read_directory(&is, buffer);

 /* Read in the second half of the image from the second imagery file */
 for (i = 0; i < (ds2.num_records_vol - 1) / 3; ++i)
 {
   lseek(fd, line_front, 1);
   read(fd, green, image_width);
   write(fdg, green, image_width);
   lseek(fd, line_back + line_front, 1);
   read(fd, red, image_width);
   write(fdr, red, image_width);
   lseek(fd, line_back + line_front, 1);
   read(fd, infrared, image_width);
   write(fdir  infrared, image_width);
   lseek(fd, line_back, 1);
 }

#else FULL_IMAGE               /* only read out 512 x 512 subset images */
  printf("line_front=%d  line_back=%d  line_length=%d image_width=%d\n",
         line_front, line_back, line_length, image_width);
  line_back += image_width - 512;
  printf("new_line_back=%d\n", line_back);

  for (i = 0; i < 512; ++i)
  {
    lseek(fd, line_front, 1);
    read(fd, green, 512);
    write(fdg, green, 512);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, red, 512);
    write(fdr, red, 512);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, infrared, 512);
    write(fdir, infrared, 512);
    lseek(fd, (long) line_back, 1);
  }
#endif FULL_IMAGE
}
```

APPENDIX D. PDEF PARSER FOR SPOT DATA.

This appendix presents a PDEF parser for SPOT data which is written using the parsing programs "lex" and "yacc." Notice how much simpler this code is than that presented in Appendix C. This code is also much more general than that found in Appendix C. To parse a different type of file only small changes need be made to the parser and a new keys.h file will have to be defined. The more PDEF parsers that you write, the easier it gets.

The lex and yacc programs began as part of the UNIX operating system, but they have proven useful enough to be written for other operating systems as well. Implementations of lex and yacc exist for MSDOS, VMS and OS/2. Other implementations probably exist. This means that this parser should be portable to other operating systems. In fact, since lex and yacc generate C programs, the results of the operations of lex and yacc can be compiled on systems on which lex and yacc themselves are not available.

```
%{
/**  LEX
This lex code parses a PDEF information file which contains information
about a raster data file.
The characters ':' and '#' are reserved to identify keys and for
in-file comments.
*/

#include <stdio.h>       /* to get EOF */
#include <math.h>        /* get type of atof() */
#include "info.h"        /* definition of cnode */

/* This is parse.tab.h if generated by bison; y.tab.h if generated by yacc */
#include "parse.tab.h"
/* #include "y.tab.h"   /* if generated by yacc */

%}

name              [A-Za-z_]?[-A-Za-z_0-9]
phrase            [A-Za-z_]?[-A-Za-z_0-9 ;=()]
integer           [-+]?[0-9]
real              [-+]?[0-9]*"."[0-9]
word              [^:#\n \t]
line              [^:#\n]
string            [^:#]
a                 [Aa]
b                 [Bb]
c                 [Cc]
d                 [Dd]
e                 [Ee]
f                 [Ff]
g                 [Gg]
h                 [Hh]
i                 [Ii]
l                 [Ll]
m                 [Mm]
n                 [Nn]
o                 [Oo]
p                 [Pp]
r                 [Rr]
s                 [Ss]
t                 [Tt]
y                 [Yy]
sp                [ ]

%START            S
%a                3000
%o                6000

%%

"#".*$            {
                          ;                       /* Comment; ignore */
                          /* printout();          /* DEBUG */
                  }
^{t}{i}{t}{l}{e}:       |
^{c}{o}{m}{m}{e}{n}{t}: |
^{l}{e}{a}{d}{e}{r}\ {n}{o}{t}{e}{s}:    |
^{i}{m}{a}{g}{e}{r}{y}\ {n}{o}{t}{e}{s}:        |
^{t}{r}{a}{i}{l}{e}{r}\ {n}{o}{t}{e}{s}:        |
^{n}{o}{t}{e}{s}:        { /* multiline fields may follow these keys */
                              yyless(yyleng - 1);   /* push back ':' */
                              yylval.string = yytext;
                              BEGIN S;        /* string mode */
                              return KEY;
                         }
^{phrase}+:        {  /* This could be {name}+: if no spaces allowed */
                        yyless(yyleng - 1);   /* push back ':' */
                        yylval.string = yytext;
                        return KEY;
                   }
{real}*            {  /* This could get a monster like "+." ; oh well. */
                        yylval.real = atof(yytext);
                        return REAL;
                   }
{integer}+         {
                        yylval.integer = atoi(yytext);
                        return INTEGER;
                   }
{word}+            {
                        yylval.string = yytext;
                        return STRING;
                   }
<S>{string}+       {
                      /*
                        For matching a multi-line string, I am using a
                        mode that is so permissive that it is also
                        matching the next key.  Since the key
                        must begin a line, I just search backwards to find
                        the last '\n' and push the part after
                        this '\n' back to the input stream.  Of course, this
                        behavior means that a 'string' may not end the
                        file being scanned; i.e. we may not be in string mode
                        at EOF.  A way to finesse this is to have the key
                        END: at the end of the file being scanned.
                       */
                        int i;

                        for (i = 0; yytext[yyleng - 1 - i] != '\n'; ++i)
                                ;
                        yyless(yyleng - 1 - i); /* push back next key */
                        yylval.string = yytext;
                        BEGIN 0;
                        return STRING;
                   }
\n                 { ; /* Prevent spurious newlines */ }
.                  { ; /* This filters out leftover separators */ }

%%

#undef yywrap

yywrap()
{
        return EOF;
}
```

```
%{
/** YACC
 * This is a yacc program to Parse a PDEF information file
 *
 * Entries in the form of 'entry' structs are formed into a list
 * which is based on the global (struct entry *) 'infolist' .
 *
 * Key phrase entries are lowercased for ease of comparison.
 */

#include <stdio.h>                /* define NULL, stderr */
#include <ctype.h>                /* isupper(), tolower(), etc. */
#include "info.h"
#include "keys.h"                 /* Key phrases are defined here */

/* Always define YYDEBUG.  Set extern int yydebug=1 to turn on debugging. */
#define YYDEBUG 1

#define NONAME ((char)0)

/*
 * This is where the item list gets built.  Initialized to 0 as an extern.
 */
struct entry *infolist;

/* Local variables */
static struct entry *item;

extern char *malloc();

%}

%union {
  int     integer;
  double  real;
  char   *string;
  int    *pixval;
  cnode  *colormap;
}

%start info

%token <string>         KEY
%token <integer>        INTEGER
%token <real>           REAL
%token <string>         STRING
%token <colormap>       COLORMAP

%type <string> key
%type <pixval> pixval
%type <colormap> cmap

%%

info    :       /* no input */
        |       info entry
;
```

```
entry   :       key INTEGER
        = { /* Don't need the "=" sign but more readable with it. */
            item->value->integer = $2;
            item->type = INTEGER;
        }
        |       key REAL
        = {
            item->value->real = $2;
            item->type = REAL;
        }
        |       key STRING
        = {
            item->value->string = malloc(strlen($2) + 1);
            strcpy(item->value->string, $2);
            item->type = STRING;
        }
        |       key cmap
        = {
            item->value->colormap = $2;
            item->type = COLORMAP;
        }
        |       key             { ; /* No associated value; Do nothing. */ }
        |       error           { ; /* Do nothing. Report error. */ }
;

key     :       KEY
        = {
                /*
                 * This must be defined as an intermediate when parsing a
                 * string data type since parsing
                 * only uses a single char buffer and parsing two strings
                 * in a row will cause the second one to overwrite the
                 * first one.  Took a long time to find this out.  Sigh.
                 * Since I had to define it for parsing strings, I may as
                 * well use it for everything.  Good place to allocate
                 * a new item struct and link it into infolist.
                 */
                if(check_key($1) == NONAME)
                {
                  fprintf(stderr, "Key  \"%s\" not found\n", $1);
                  YYERROR;  /* Cannot find key; declare an error. */
                }
                else
                {
                  alloc_item($1);
                }
        }
;

cmap    :       pixval
        = {
                $$ = (cnode *)malloc(sizeof(cnode));
                $$->pixval = $1;
                $$->next = (cnode *)NULL;
        }
        |       cmap pixval
        = {
                $$ = (cnode *)malloc(sizeof(cnode));
                $$->pixval = $2;
                $$->next = $1;
        }
;

pixval  :       INTEGER INTEGER INTEGER INTEGER
        = {
                $$ = (int *)malloc(4 * sizeof(int));
                $$[0] = $1;
                $$[1] = $2;
                $$[2] = $3;
                $$[3] = $4;
        }
;

%%

check_key(key)
char *key;
{
  int i;

  /* Destructively lowercase key */
  for (i = 0; i < strlen(key); ++i)
    key[i] = isupper(key[i]) ? tolower(key[i]) : key[i];

  for (i = 0; *keys[i] != NONAME; ++i)
    if (strcmp(keys[i], key) == 0)
      return (int)key;

  return NONAME;
  }

alloc_item(key)
char *key;
{
  item = (struct entry *)malloc(sizeof(struct entry));
  item->key = malloc(strlen(key) + 1);
  strcpy(item->key, key);

  item->value = (vals *)malloc(sizeof(vals));
  item->type = 0;               /* no defined type yet */

  /* link the item into the item list */
  item->next = infolist;
  infolist = item;
}

yyerror(s)
char *s;
{
        perror(s);
}
```

```
/*
 * A main program to test parsing of PDEF information files.
 * This code prints out the various fields and data types defined.
 */
#include <stdio.h>
#include "info.h"              /* Locally defined types, etc. */
#include "parse.tab.h"         /* bison/yacc defined types */

main()
{
  int     i;
  int     error;
  cnode   *cmap;
  struct entry *current;
  cnode   *reverse_colormap();

  /* yydebug = 1;                      /* debug mode if 1 */
  error = yyparse();
  if (error != 0)
    yyerror("Parse Error returned by yyparse() to main(): ");
  else {
    printf("\n\nEntering printout of parsed infolist:\n\n");
    for (current = infolist; current != NULL; current = current->next) {
      switch (current->type) {
      case 0:                  /* no value associated with key */
        printf("%s:\n", current->key);
        break;
      case INTEGER:            /* integer types */
        printf("%s:\tINTEGER %d\n",
               current->key, current->value->integer);
        break;
      case REAL:               /* floating-point number types */
        printf("%s:\tREAL %lf\n",
               current->key, current->value->real);
        break;
      case STRING:             /* string types */
        printf("%s:\tSTRING %s\n",
               current->key, current->value->string);
        break;
      case COLORMAP:           /* colormap type (int **) */
        printf("%s:\n", current->key);
        printf("     pixel       red     green      blue\n");
        printf("     -----       ---     -----      ----\n");

        for (cmap = current->value->colormap; cmap != NULL;
             cmap = cmap->next) {
          printf("%10d%10d%10d%10d\n",
                 cmap->pixval[0], cmap->pixval[1],
                 cmap->pixval[2], cmap->pixval[3]);
        }
        break;
      default:
        fprintf(stderr, "Cannot find type %d\n", current->type);
      }
    }
    printf("\nAll done.  Bye now.\n");
  }
}
```

APPENDIX E. PDEF PROGRAM TO USE PARSED SPOT DATA.

This appendix shows how programs can use a PDEF file to read and work with SPOT data. The program listed here uses a PDEF file to make separate red, green, and blue images from the SPOT band-interleaved image file in its distributed format.

```
/*
 * This program uses a PDEF information file to read SPOT data in and
 * writes out separate arrays for the green, red, and infrared bands.
 * This only works for multiband data in band interleaved format.
 */

#include <stdio.h>
#include <sys/file.h>
#include "info.h"                   /* Locally defined types, etc. */
#include "parse.tab.h"              /* bison/yacc defined types */

#define LINE_HEADER 32
#define LINE_TRAILER 68
#define TRUE 1
#define FALSE 0
#define FULL_IMAGE                  /* undefine this to get 512x512 images */

main(argc, argv)
  int     argc;
  char   **argv;
{
  char    *buffer;
  char    *green, *red, *infrared;         /* unpacked image line buffers */
  struct entry *current;            /* used to read the parse list */
  int     i;
  int     fd,                       /* general-purpose file descriptor */
          fdg, fdr, fdir,           /* file descriptors for unpacked data */
          line_length1,             /* number of bytes per imagery record */
          line_length2,
          number_records1,          /* number of records in image files */
          number_records2,
          line_front,               /* number of junk pixels at line start */
          line_back;                /* number of junk pixels at line end */
  int     image_width;              /* number of image pixels per line */
  int     error;
  int     image = FALSE;            /* Boolean to see if image data */
  extern char *malloc();
  extern char *realloc();

  if (argc < 3)
    fprintf(stderr, "Usage: %s imagery1 imagery2  < info_file\n", argv[0]), exit

  /* yydebug = 1;                           /* debug mode if 1 */
  error = yyparse();
  if (error != 0)
  {
    yyerror("Parse Error returned by yyparse() to main(): ");
    exit(1);
  }

  /* Get the information needed from the parsed infolist.  */
  for (current = infolist; current != NULL; current = current->next)
  {
    /* Get info for first imagery file */
    if (strcmp("imagery length of largest record in file", current->key)
        == 0)
      line_length2 = line_length1 = current->value->integer;
    if (strcmp("imagery number of records in this volume", current->key)
        == 0)
      number_records2 = number_records1 = current->value->integer;
#ifdef FOO  /* Both images are the same anyway. */
    /* Get info for second image file */
    if (strcmp("length of largest record in file", current->key) == 0)
      line_length2 = current->value->integer;
    if (strcmp("number of records in this volume", current->key) == 0)
      number_records2 = current->value->integer;
#endif
    /* Get info originally from image header */
    if (strcmp("left border pixels per line", current->key) == 0)
      line_front = LINE_HEADER + current->value->integer;
    if (strcmp("right border pixels per line", current->key) == 0)
      line_back = current->value->integer + LINE_TRAILER;
    if (strcmp("image pixels per line", current->key) == 0)
      image_width = current->value->integer;
  }

  /* Open and seek past the first record of the first imagery file */
  buffer = realloc(buffer, line_length1);
  if ((fd = open(argv[1], O_RDONLY, 0775)) <= 0)
    perror("open imagery failed"), exit(1);
  if (lseek(fd, line_length1, 0) == -1)
    perror("lseek imagery failed"), exit(1);

  /* Open files to receive the unpacked image as three color arrays */
  if ((fdg = open("green", O_WRONLY | O_CREAT, 0775)) <= 0)
    perror("open green failed"), exit(1);
  if ((fdr = open("red", O_WRONLY | O_CREAT, 0775)) <= 0)
    perror("open red failed"), exit(1);
  if ((fdir = open("infrared", O_WRONLY | O_CREAT, 0775)) <= 0)
    perror("open infrared failed"), exit(1);

  green = malloc(image_width);
  red = malloc(image_width);
  infrared = malloc(image_width);

  printf("line_front=%d  line_back=%d  line_length=%d\n image_width=%d number_re
         line_front, line_back, line_length2, image_width, number_records2);

#ifdef FULL_IMAGE

  /*
   * Read in the first half of the image from the first imagery file. We
   * are currently at the beginning of the first image data line.
   */
  for (i = 1; i <= number_records1 / 3; ++i)
  {
    lseek(fd, (long) line_front, 1);
    read(fd, green, image_width);
    write(fdg, green, image_width);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, red, image_width);
    write(fdr, red, image_width);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, infrared, image_width);
    write(fdir, infrared, image_width);
    lseek(fd, (long) line_back, 1);
  }
  printf("Image file #1 done: Last record read was %d\n", --i * 3 + 1);

  /* Open and read the first record of the second imagery file */
  buffer = realloc(buffer, line_length2);
  if ((fd = open(argv[2], O_RDONLY)) == 0)
    perror("open imagery failed"), exit(1);
  if (lseek(fd, line_length2, 0) == -1)
    perror("lseek imagery2 failed"), exit(1);

  /* Read in the second half of the image from the second imagery file */
  for (i = 1; i <= number_records2 / 3; ++i)
  {
    lseek(fd, line_front, 1);
    read(fd, green, image_width);
    write(fdg, green, image_width);
    lseek(fd, line_back + line_front, 1);
    read(fd, red, image_width);
    write(fdr, red, image_width);
    lseek(fd, line_back + line_front, 1);
    read(fd, infrared, image_width);
    write(fdir, infrared, image_width);
    lseek(fd, line_back, 1);
  }

#else FULL_IMAGE                /* only read out 512 x 512 subset images */
  line_back += image_width - 512;
  printf("new_line_back=%d\n", line_back);

  for (i = 0; i < 512; ++i)
  {
    lseek(fd, (long) line_front, 1);
    read(fd, green, 512);
    write(fdg, green, 512);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, red, 512);
    write(fdr, red, 512);
    lseek(fd, (long) (line_back + line_front), 1);
    read(fd, infrared, 512);
    write(fdir, infrared, 512);
    lseek(fd, (long) line_back, 1);
  }
#endif FULL_IMAGE

}
```

APPENDIX F. STORING VECTOR AND POLYGON DATA USING PDEF.

This appendix shows how programs can use a PDEF file to read and work with vector data. There are no programs here, but rather a schematic design of a file structure suitable for handling vector data. The information about what should go in these files is taken from a United States Geological Survey (USGS) publication.[1] This document details the format used for a GIS called the Geographic Information Retrieval and Analysis System (GIRAS).

GIRAS has some quaint archaisms, such as using 80-character card images and EBCDIC character encoding, but the basic needs of polygon data are taken care of in GIRAS, and these needs guided the design of the format outlined in this appendix.

A GIRAS *map* is divided into *sections* which are treated separately, as they will be in this design. For PDEF, each GIRAS map will be in its own directory. Following is a schematic of the directory structure. Directories are capitalized and files have lowercase names. A description of each of the files mentioned below will follow in the body of the text:

```
Map Directory:
        map.info
        attribute.info
        Section Directory:    (may be several of these directories)
                section.info
                node.info
                node.data
                arc.info
                arc.index
                arc.data
                polygon.info
                polygon.index
                polygon.data
```

All files that have the suffix ".info" are PDEF information files; all other files contain binary data that is accessed using the descriptions in the information files. These names are conventional and not enforced by PDEF. A map directory contains all information about a GIRAS map. Nodes, arcs, and polygons are stored in separate files. Arc files refer back to nodes and polygon files refer back to arcs.

A node is stored as a single x,y point with its associated node ID number. Arcs are stored as a series of x,y points. Nodes at the ends of arcs are referenced to the node file. Polygons are stored by referring to the arcs of which they are composed. So to read the data for a polygon, arc data and node data must be accessed as well. Attributes are associated with polygons and also with arcs. This is redundant. However, by having attributes attached to arcs as well as polygons, the

[1] United States Geological Survey, "Land Use and Land Cover Digital Data from 1:250,000- and 1:100,000-Scale Maps." USGS Data Users Guide no. 4.

validity of a polygon can be tested after it is assembled from its arcs; all attribures should check. Encoding attributes in arcs also allows an arc to form a linear feature independent of a polygon. For a linear feature, both arc attributes are the same. A description of each of the file types follows:

map.info

The map.info file contains global information about the map such as any codes and catalog numbers, map projections and scales, source information and dates, quality and tolerance, and other housekeeping data. This kind of data lends itself very well to a PDEF information file format.

attribute.info

This is an attribute table assigning attribute numbers (also called "codes" or "indexes") to their meanings. This is done by matching numbers to text strings. The text strings define the meanings of the attributes. If attribute codes are invariant, this file could actually be at the topmost directory level and apply to all map files under it.

Section Directory

To keep index numbers and coordinate numbers small, data is segmented into sections. A directory is defined for each section under a given map. The structure of each of these directories is the same. The rest of the files described below are in a section directory.

section.info

This file gives the glob l information used in this section. This includes the bounding coordinates of the section, a section number, processing history, and other cataloging data.

node.info

A declaration of the data structure used to encode a node is in here. Nodes have an x,y coordinate and a node index number, so this file may look like this:

number of bytes for x coordinate: 2
number of bytes for y coordinate: 2
number of bytes for node index number: 4

Nodes are then read from the node.data file using this declared structure, i.e. the offset from one node to the next is 8 bytes.

node.data

This is a binary file of node data as described in the node.info file.

arc.info

This is a PDEF information file describing the format of data in the binary files arc.index and arc.data.

arc.index

This is a binary file of data giving information about arcs whose coordinates are stored in arc.data. Formatting of this file is specified in arc.info. This file contains offsets and point counts needed to read data from arc.data. It also contains other information about arcs such as the node numbers of the beginning and ending node. Each arc has a fixed-length record in this file. An example of the contents of such a record is:

arc index number
beginning node index number
ending node index number
number of points
iata file offset to first point
left attribute
right attribute
left polygon
right polygon
...

arc.data

This is a binary file consisting of nothing but sequences of x,y points. The length of each coordinate in bytes is specified in the arc.info file.

polygon.info

This is a PDEF inforr. .ion file describing the format of data in the binary files polygon.index and polygon.data. Comments and other global data, such as the total number of polygons in this section, are put in here as well.

polygon.index

This is a binary file of data giving information about polygons whose arc index numbers are stored in polygon.data. Formatting of this file is specified in polygon.info. This file contains offsets and point counts needed to read data from polygon.data. It also contains other information about polygons, such as the associated attribute index. Each polygon has a fixed-length record in this file. An example of the contents of such a record is:

polygon index number
number of arcs
data file offset to first arc index
attribute
...

polygon.data

This binary file contains arc indices as signed integers. A negative arc number here means that the arc is to be traversed in a reverse direction in order to traverse the polygon in a positive (clockwise) sense.